Scheme Programming (cnt.)

- Variable Bindings
- Higher-order Programming & Anonymous Functions
- Unrestricted Functions

Variable Bindings

Which of the following expressions is necessarily a *let∗*?
What is the returned value in each case?

```
 (let ((sqr (lambda (x) (* x x)))
       (x 3))
    (sqr x))
–> ????
(lett ((x 3))
       (sqr (lambda (x) (* x x)))
    (sqr x))
–> ????
```

Response: None. Why? Because, variable "x" is **local** to the scopes of the λ-expressions. Both return **9**.

Variable Bindings (cnt.)

And on the following expression, shall I use *let*, *let∗*, or *letrec*?

```
 (let ((f (lambda (x) (g 1 x)))
       (g (lambda (a x) (+ x a)))
       (x 3))
    (f x))
–>???
```

If I use *let∗*, Scheme will return an error, because I am calling a function, **g**, which has not been defined. Therefore, I have to use *letrec*, since Scheme will **fully expand** the body of the first λ-expression, before assigning it to the variable f.
The returned value is 4.

Rewriting functions in terms of **reduce**

```
(define (reduce f L e)
   (if (null? L) e
        (f (car L) (reduce f (cdr L) e))))
```

Anexample with **reduce**

;**Purpose**: this function takes as argument a list of numbers and
; returns a list whose unique element is the *max* of the input list
;**Assumptions**: L is a non-empty list
;**Auxiliary functions called:** none

```
(define (max L)
   (cond ((null? (cdr L)) (car L))
      ((> (car L) (cadr L)) (max (cons (car L) (cddr L))))
      (else (max (cdr L)))))
```

Writing *and* in terms of **reduce**

```
(define (and L)
   (if (car L) (and (cdr L))
       #f))
```

Motivated by: (reduce and '(2 #f 1) #t)

$$\underbrace{(and\ 2\ \underbrace{(and\ \#f\ \underbrace{(and\ 1\ \#t)}_{\#t})}_{\#f})}_{\#f}$$

Rewriting *and* to use it with *reduce*:

```
(define (and x y)
   (if x L
       #f))
```


Writing *max* in terms of **reduce**

Motivated by: (reduce max '(2 3 1) 2)

$$\underbrace{(max\ 2\ \underbrace{(max\ 3\ \underbrace{(max\ 1\ 2)}_{2})}_{3})}_{3}$$

Rewriting *max* to use it with *reduce*:

```
(define (max x y)
   (if (> x y) x
       y))
```


Unrestricted Functions

Now implementing a function *myand* that takes any number of arguments and makes an *and* with these arguments. E.g.:
(myand 3 4 'a #f 'r #t) –> #f
(myand 4 'd 2 #t 6) –> #t
(myand) –> #t

```
(define myand
  (letrec ((list-and
    (lambda (L)
       (cond ((null? L) #t)
             ((car L) (list-and (cdr L)))
             (else #f)))))
  (lambda args (list-and args))))
```